Summer Fellowship Report

Spoken Tutorial Event Logs and Analytics System

By:-

Krithik Vaidya,

B. Tech, IInd Year (Information Technology)

NITK Surathkal

Table of Contents

Introduction

This is the project report for my work done as a FOSSEE Summer Fellow under the Spoken Tutorials project. My work involved creating pipelines for extracting user activity logs (referred to as *Event Logging*) as users browse and interact with the website, and storing them into a MongoDB database, thus enabling data analysis and visualization. The analytics system for analyzing these logs was also parallelly created by my teammate.

The Event Logging system consists of two main parts:-

- Saving of user activity logs as they browse and visit different parts of the website (henceforth referred to as *Website Logs*).
- Saving and display of course completion and tutorial progress statistics for users (referred to as *Tutorial Progress Logs*)

The open-source code for the project can be found on GitHub -

spoken-website (branch: logs-krithik)
spoken-website (branch: logs-krithik-js)
Spoken-Analytics-System (branch: master)

Presentation Video

I. Website Logs

Our first goal was the saving of user activity logs for all the pages on the website. For each visit to a page on the Spoken Tutorials website, we are currently logging information such as:

1. Username of visitor
2. Path of web page visited
3. Browser info, Operating System info, and Device info
4. IP address
5. Date and Time of visit (in UTC timezone)
6. Whether the visit was first time or returning
7. Latitude and Longitude of user
8. Country, Region, and City from which the request originated, if available
9. Request method (GET, POST, etc.)
10. Referring link
11. Page title
12. Arguments and Keyword Arguments sent to the view in the request, if any.
13. POST request data, if any.

Two different architectures have been created for extracting and saving Website Logs. Both architectures are soon talked about in detail:-

- Django Middleware-based approach (server-side)
- Javascript-based approach (client-side)

Both these systems have their own pros and cons, which are detailed at a later point below.

Next, the final architectures of both the above approaches will be discussed, with the reasonings for the chosen architectures talked about after.

Final architecture, Middleware-based Approach

This approach firstly consists of extracting all the log data in the Django middleware, except for the location data. After extracting the data in the middleware, we make an asynchronous call to an API function, created in the Spoken-Analytics-System (in *logs_api/views.py,* the *save_middleware_log()* function view). This function will get the location data, and then enqueue the log in a Redis queue named *middleware_log*.

For obtaining the location information, we will perform IP-based Geolocation - i.e., we will map the IP address of the visit to location data using the GeoIP2 Geolocation service.

A separate monitoring program has been written (*monitor_queue.py*) to monitor the Redis queue. When the number of items in the queue reaches ≥ n (where n is defined by the MONGO_BULK_INSERT_COUNT variable in settings.py), the first n items will be extracted from the queue. According to the value of SAVE_LOGS_WITH_CELERY (True/False, defined in settings.py), the extracted logs will either be saved in bulk directly, by the *monitor_queue()* function itself, or sent as a task to Celery to be asynchronously executed.

The saving of the logs in MongoDB is done using the pymongo library. The extraction of Brower info, OS info, and Device info in the Django middleware is done by using the django-user-agents library.

To record first-time visits, we set the option SESSION_EXPIRE_AT_BROWSER_CLOSE in Django settings to False, and then use the following snippet:

```
if 'has_visited' in request.session:
    data['first_time_visit'] = "false"

else:
    data['first_time_visit'] = "true"
    request.session['has_visited'] = True

request.session.set_expiry(15552000)  # 6 months, in seconds
```

The HTML title of the webpage is not available on the server-side, but is required for visualization purposes. To overcome this, we have a separate *event_name* field in the logs. The event name is determined by Regex matching the URL to a predefined list of patterns of URLs defined by us. After the match is found, it will use the corresponding event name. The EVENT_NAME_DICT used for matching can be found in the spoken-website repo (see below for the screenshot), *logs/urls_to_events.py* file. The drawback is that for every new URL added to the spoken website that does not match a previously defined pattern, the dict needs to be updated too. Else the logs for visits to those URL patterns won't be recorded.

```
EVENT_NAME_DICT={
    # maps root
    r'^/$' : {
        'name' : 'event.home.view'
    },

    r'^/home/$' : {
        'name' : 'event.home.view'
    },

    r'^/watch/([0-9a-zA-Z-+%\(\)., ]+)/([0-9a-zA-Z-+%\(\)., ]+)/([a-zA-Z-]+)/$' : {
        'name' : 'event.video.watch'
    },

    r'^/cdcontent/$' : {
        'name' : 'event.cdcontent.download'
    },

    r'^/tutorial-search/$' : {
        'name' : 'event.tutorial.search'
    },
```

The above is a part of the dict

The logs stored in MongoDB can then be used for performing analysis and creating visualizations.

The branch containing the code for this approach can be found here (the branch also contains the implementation of the Tutorial Progress Logging system). The code written for the API can be found in the Spoken-Analytics-System repository.

Explanation of choices in the architecture

Usage of API for saving logs, instead of doing it locally

Initially, the middleware asynchronously called a local function (i.e. running on the same system as the server) to do the Geolocation and the pushing of logs to Redis. The Redis monitoring consumer function was also running locally, in the same system as the spoken-website. However, we decide to scrap this approach and transfer the function to a separate API in the Spoken-Analytics-System because:

- It is easier to perform an asynchronous HTTP call, rather than calling a local function asynchronously. Calling a local function asynchronously complicates the code by introducing asyncio and multithreading.
- In case of high loads on the spoken-website server, the system and server would be slowed down.
- Since there are a fewer changes to the main spoken website repository, it would be quicker and easier to review and deploy to production.
- It makes sense to have the Event Logging and Analytics related code under a separate subsystem.

Asynchronous saving of logs in middleware implementation

We could have directly done Geolocation and saved the log into MongoDB in the middleware itself, without needing to involve Redis queues and separate APIs. However, these are blocking operations that would increase the page load times for the user especially in case of high load scenarios. It also has the drawbacks mentioned in the previous section, related to saving logs locally instead of API based approach. Also, reliable bulk saving of logs would have been difficult to achieve using a purely middleware-based approach.

Giving the option of using Celery for bulk saving of logs

Celery is a full-featured task processing software for Python web applications used to asynchronously execute work outside the HTTP request-response cycle. It provides features such as automatic scaling of the number of workers, multiprocessing to perform

concurrent execution of tasks, etc. There are applications like Flower to monitor Celery workers and the task queues. A Celery system can consist of multiple workers and brokers, giving way to high availability and horizontal scaling.

Hence Celery is also a good option for running tasks asynchronously.

Usage of pymongo for saving logs into MongoDB

To save the logs in MongoDB from a Python function, we started by using pymongo. Pymongo simply takes a Python dictionary and saves it as a document in the specified MongoDB database and collection. Since there was no validation being done, we decided to add validation to the MongoDB database. This is the current validation schema, after multiple iterations of refinements:

https://gist.github.com/krithikvaidya/18bcaba3d9a87d8a40e92c4cdd1bac95

The above is the MongoDB validation schema for the website event logs collection. This is to be entered in the mongo shell, after selecting the appropriate database. The above deals with validating the data just before the document enters the database. This is not necessary to do, but will be helpful in ensuring consistency.

For tighter coupling between our Django application and the MongoDB database, we decided to try replacing the use of pymongo with Djongo. Djongo is a SQL to MongoDB query transpiler. It translates a SQL query string into a MongoDB query document. As a result, all the Django ORM features, work as-is. It would also let us interact with the Djongo model through the Django admin page.

Hence, using Djongo would add another layer of validation and consistency to the log data in the MongoDB database. Using Djongo allows you to interact with MongoDB exactly as you'd interact with SQL databases using models, with some additional MongoDB specific features.

However, the load testing (talked about below) showed that pymongo performed many orders of magnitude faster than djongo.

Choosing the right architecture by performing Load Testing

(This load testing was done before the API based implementation of Middleware-based logs).

Till now, a single async function task/celery task dealt with the storing of a single log in the database. To possibly speed up the saving of logs, we considered bulk saving of logs in a single task (for e.g. 1000 logs in 1 task) instead of a single log per task.

To determine the best setup amongst the different setup described previously, I proceeded to do extensive load testing - using different combinations of

- Celery/our local async function
- Djongo/pymongo
- Single log per task/multiple logs per task.

Testing method: Queue with 10000 tasks for each of the (2 * 2 * 2) = 8 setups.

Testing environment - my local machine.

According to the load testing observations,

- The setups using pymongo performed many orders of magnitude better than those using Djongo. This is probably because of the extra Djongo model layer between Django and MongoDB. For every log that needed to be saved, an object of the model had to be created, and the validations, etc. brought in a large overhead. The feature benefits provided by Djongo over pymongo was heavily outweighed by its lower performance and the additional complexity of usage.
- The setups implementing Bulk saving of logs performed much better than single saving. This is because the time required for inserting a log is determined by the following factors, where the numbers indicate approximate proportions:
    - Connecting: (3)
    - Sending query to server: (2)
    - Parsing query: (2)

    - Inserting logs: (size of log)
    - Closing: (1)

  Obviously, bulk saving of logs will be much faster since only a single connection is opened for multiple logs.

- In this environment, our own consumer function's performance was better than Celery's. But, Celery is a much more full-featured task processing software, as talked about previously.

So, considering all the factors, we decided to choose the setup with

- an option to either use Celery or saving the logs in monitor_queue.py itself,
- pymongo for saving of logs into MongoDB, and
- bulk saving of logs per task.

There is a SAVE_LOGS_WITH_CELERY setting in the settings.py file. Setting it to true will use Celery as the task processing queue to save the logs, and setting it to false with use the separate queue monitoring Python function (defined in monitor_queue.py) as the task processing queue. The MONGO_BULK_INSERT_COUNT value controls how many logs are inserted, per bulk insert operation.

The monitor_queue.py script

The *monitor_queue.py* script (found in the Spoken-Analytics-System repo) monitors the Redis queue, whose name is decided by the value of the USE_MIDDLEWARE_LOGS setting. The number of logs it extracts from the Redis queue per bulk insertion is determined by the MONGO_BULK_INSERT_COUNT settings variable. Many precautions have been taken to avoid crashes. It uses Python's in-built logging system to periodically print out informative log messages. A sample run of the script is as below:

```
(spoken-analytics-system) krithik@krithik-Strix-15-GL503GE:~/Desktop/Git/Spoken-Analytics-System$ python monitor_queue.py
[monitor_queue] [2020-06-07 09:55:55,982] ERROR : Either the redis server is not running, or the redis configurations are incorrect.
[monitor_queue] [2020-06-07 09:56:00,984] INFO : Successfully connected to Redis server.
[monitor_queue] [2020-06-07 09:56:00,985] INFO : Number of items in queue middleware_log: 0
[monitor_queue] [2020-06-07 09:56:05,988] INFO : Number of items in queue middleware_log: 26
[monitor_queue] [2020-06-07 09:56:10,992] INFO : Number of items in queue middleware_log: 59
[monitor_queue] [2020-06-07 09:56:15,996] INFO : Number of items in queue middleware_log: 93
[monitor_queue] [2020-06-07 09:56:21,011] INFO : 100 items successfully inserted into MongoDB in 0.009830000000000005 seconds.
[monitor_queue] [2020-06-07 09:56:21,012] INFO : Number of items in queue middleware_log: 23
[monitor_queue] [2020-06-07 09:56:26,016] INFO : Number of items in queue middleware_log: 56
```

(Each of the iterations producing a line of output in the above has a 5-second delay. This delay can be changed by tweaking the value of the MONITOR_QUEUE_ITERATION_DELAY setting).

The code in the script has been sufficiently commented so as to be understandable.

The end result - a log stored in MongoDB


```
{
  "_id": "5ed618954901d935739f33b9",
  "path_info": "/",
  "event_name": "event.home.view",
  "visited_by": "AnonymousUser",
  "ip_address": "27.61.140.192",
  "method": "GET",
  "datetime": "2020-06-02T09:14:57.237Z",
  "referer": "(No referring link)",
  "page_title": "",
  "browser_family": "Edge Mobile",
  "browser_version": "45.3.4",
  "os_family": "Android",
  "os_version": "9",
  "device_family": "Asus X00TD",
  "device_type": "Mobile",
  "first_time_visit": "true",
  "region": "Gujarat",
  "latitude": 21.7899,
  "longitude": 73.5651,
  "country": "India",
  "city": "Rajpipla"
}
```

(In the middleware implementation, it is not possible to get the page title. However, it has been kept in the logs as an empty field for consistency)

Final Architecture, JS-based Approach

This approach consists of extracting all the log data in the client-side, using Javascript. Once the page DOM Content Loading completes, we run a function to extract all the information. To get accurate location data, we use the HTML5 Geolocation API. If the user accepts, the accurate latitude and longitude data is extracted. Else if the user declines, we query the freegeoip.app API, to map the user IP address to a location.

To get the browser and OS info, we use the browser-report JS library. To detect the device type, we use the device-detector JS library.

Once all the data is extracted, a procedure similar to that in the middleware-based architecture is used. We make an AJAX call to another API function created in the Spoken-Analytics-System (in *logs_api/views.py, the save_js_log()* view function). This view enqueues the log in a Redis queue named *js_log*. The name of the Redis queue monitored by *monitor_queue.py* (middleware_log or js_log) is decided by the value of USE_MIDDLEWARE_LOGS in settings.py file of Spoken-Analytics-System. The rest of the procedure is the same as in the middleware-based approach, involving bulk inserting logs into the DB in the monitor_queue function itself, or sending it as a task to be processed by Celery.

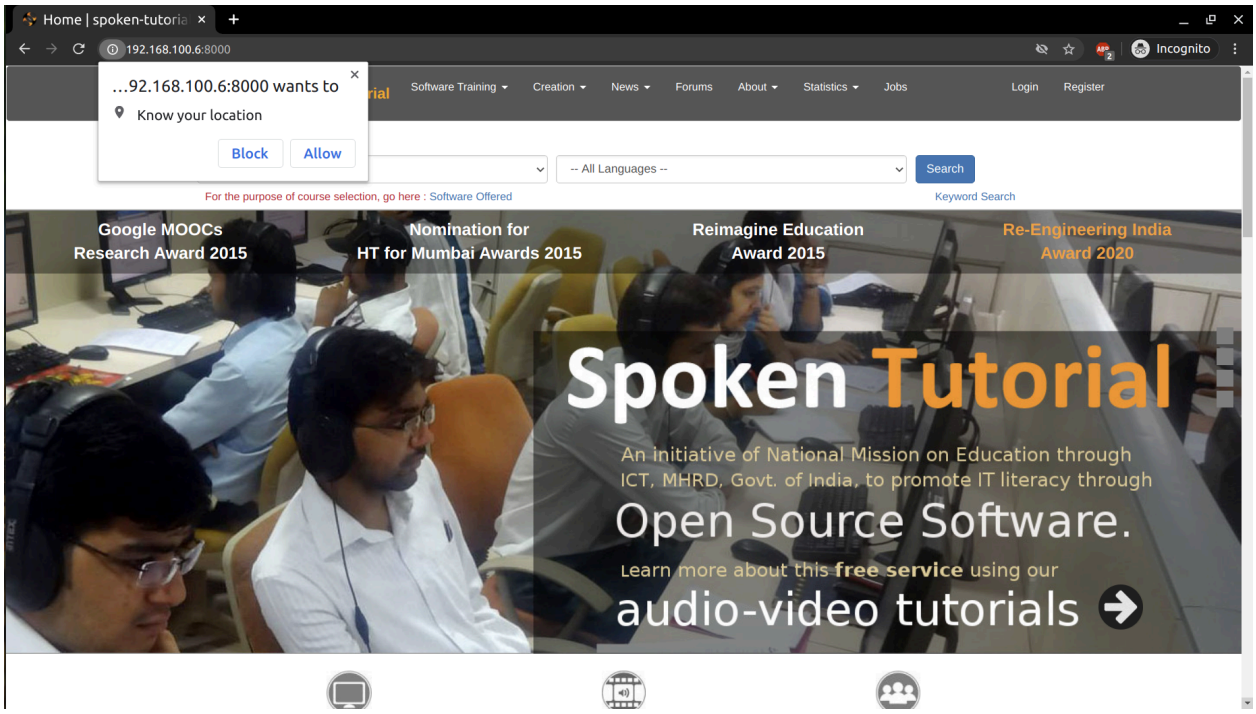
In the JS implementation, the exit link activity can also be recorded (explained below)

The JS implementation is simpler to deploy to production, as there is comparatively little extra code on the server-side. In the JS implementation, the *logs* app is only used for defining a context processor (for accessing IP address and Logs API URL) and a template tag. The middleware, views, etc are not used.

The branch containing the code for this approach can be found here (the branch also contains the implementation of the Tutorial Progress Logging system), as well as in the Spoken-Analytics-System repository.

Getting the location data in client-side JS

We can leverage the HTML5 Geolocation API to get the accurate location data of the client. However, this is subject to the user agreeing to provide their location information. The browser will put up a prompt like this when the page loads:



If the user chooses to accept, the accurate coordinates can be obtained. However, to map the coordinates to Country/Region/City data (called reverse geocoding), we will need to make an API call. There are two free APIs available for this purpose:

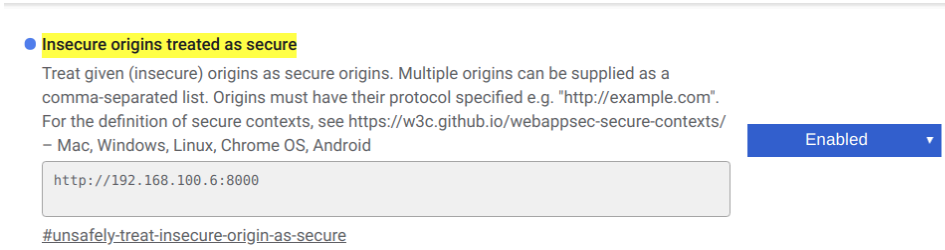
1. OpenStreetMap (Nominatim)
2. Big Data Cloud API

Although both APIs work well, they do not return data in a consistent manner (may not return region/city data, or may return it but label the fields with some other name). Hence they cannot be used in our code. Another option is to do the reverse geocoding on the

server-side using the reverse_geocoder library. However, it is somewhat slow and may bottleneck the system in case of a large load of requests.

In case the user does not accept the browser's request for location, then the freegeoip.app API will be used for the purpose of performing geolocation (which uses the IP address). This is, however, less accurate.

Side note: Modern browsers do not let user's location data be accessed using the HTML5 Geolocation API over HTTP. It is allowed only over HTTPS. In a local development environment, since the server runs on HTTP, we will need to add the IP address of our local development server to the chrome flag
chrome://flags/#unsafely-treat-insecure-origin-as-secure



Exit Link Activity

Exit links are links on the website which point to URLs having a different hostname than spoken-tutorial.org. To record exit link activity, we add an onclick event listener to all the links on a page. When a link is clicked, the Javascript checks if it has a different hostname. If it does, an AJAX call is made with the data containing exit link, page on which the link was clicked and datetime of the click, to an API on the server-side. The API saves the log into a MongoDB collection named *exit_link_logs.*

Comparison of the two Architectures

Logging with Middleware approach	Logging with Client Side JS
Recording of logs independent of client-side JS.	If the user disables JS, logs won't be recorded
No Web API calls involved, only a (very fast) call to a local geolocation database	Web API calls for Geolocation (in some cases), which may have an unpredictable latency delay.
The saving of these logs cannot be blocked by extensions on the user side.	Extensions like uBlock origin block the calls to the IP geolocation APIs, thus the logs will not be saved. (It even blocks StatCounter API from being called)
Stress on the server-side to extract all info from the request object and do geolocation	The stress of extracting info from the request object and doing geolocation is lifted from the server and done by the client side JS. The stress of maintaining an API for listening to logs sent by clients, enqueueing these logs into a redis queue, monitoring the queue and performing a bulk insert when the queue is large enough is still present with the server.
No possibility of loss of logs	Possibility for loss of logs if the user leaves the page before all the data extraction and third party API calls are complete. Since we are using Web APIs, latency is on the higher side for the API calls (usually hundreds of milliseconds)
Can't track page title. exit link data, visit length	Can track page title, exit link data, visit length.
Location accuracy restricted to that provided by GeoLite2 database	More accurate location data if the user agrees to share their location from the HTML5 geolocation API
Can access POST data	Can't access POST request data - download related logs can't be created directly
All types of HTTP requests are logged.	Only GET requests made through the browser are logged. GET requests made through other means and other types of requests are not logged.
Will possibly take a longer time to review and use in production	Will possibly take a shorter time to review and use in production

All of the above dealt with saving the website user event logs. The second part of the project deals with Course and Tutorial Progress logging.

Course and Tutorial Progress Logging

Here, we aim to log tutorial-related data while users watch the Spoken Tutorials. Using these logs, we can accomplish a lot - the user can see their course progress, see which tutorials they have completed and which they have yet to complete, continue watching at the timestamp they stopped watching the tutorial at, etc. For the analytics side, we can see which course is more popular, which tutorial is rewatched the most number of times, the average number of visits a user makes to a tutorial, etc.

Saving video logs in real-time

On the tutorial watch pages (for e.g., this one), the videos are played using the Video.js library. Video.js provides functionality to execute some logic every time the video's timestamp is updated. Using this feature, every time the minute of the video changes from the previously saved log's minute, we can make an asynchronous AJAX call to an API defined by us, to save the tutorial progress. This API will collect the data sent by the AJAX call, such as:

- The username of the tutorial watcher
- The current video time
- The current datetime
- The FOSS, the language and the tutorial,
- The number of times the user has visited that tutorial in that language
- The total length of the video.

The called API function then does some calculations to check if the video has been completed (if 80% of the minutes of the video has been crossed), etc. and updates the MongoDB document for that user.

Once this entire API call finishes successful execution, another AJAX call is made to an API to check if the tutorial should be marked as complete (if it is not already marked as

complete). According to the response, the relevant parts of the DOM are updated to display the completion status.

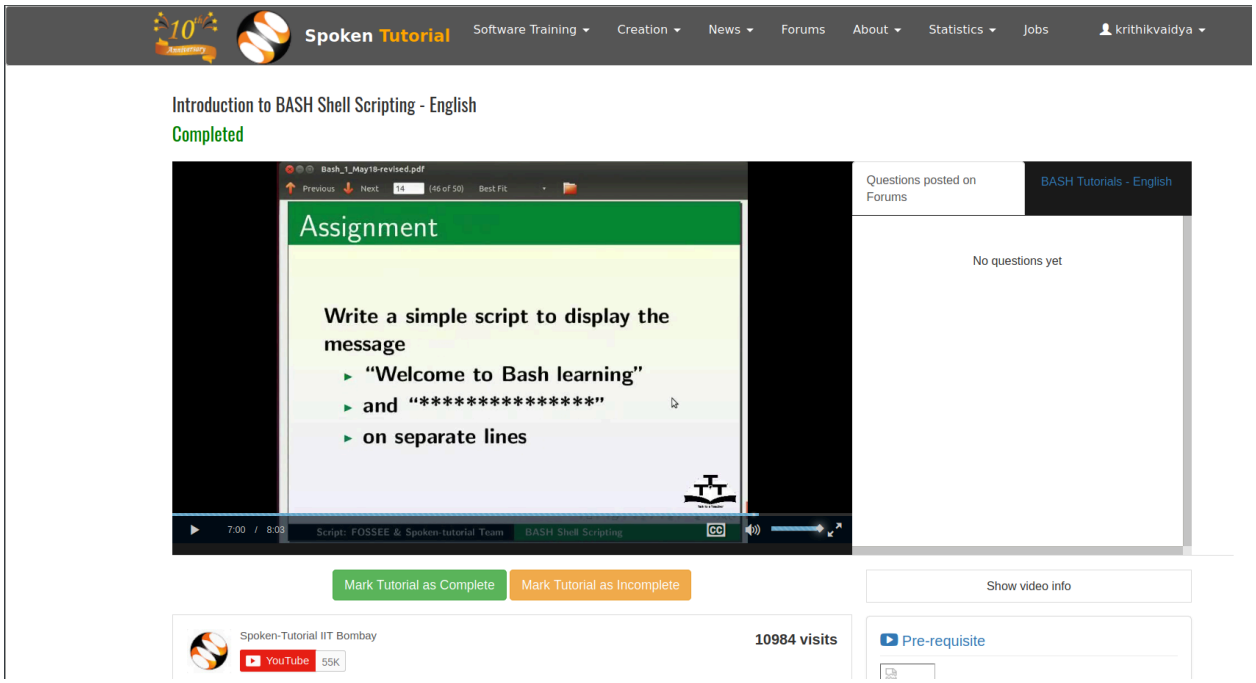
Buttons have been provided for users to mark the tutorial as completed or incomplete. Clicking the button makes another AJAX call.

For non-authenticated users, the website does not display any progress/completion data, and does not make any of these AJAX calls.

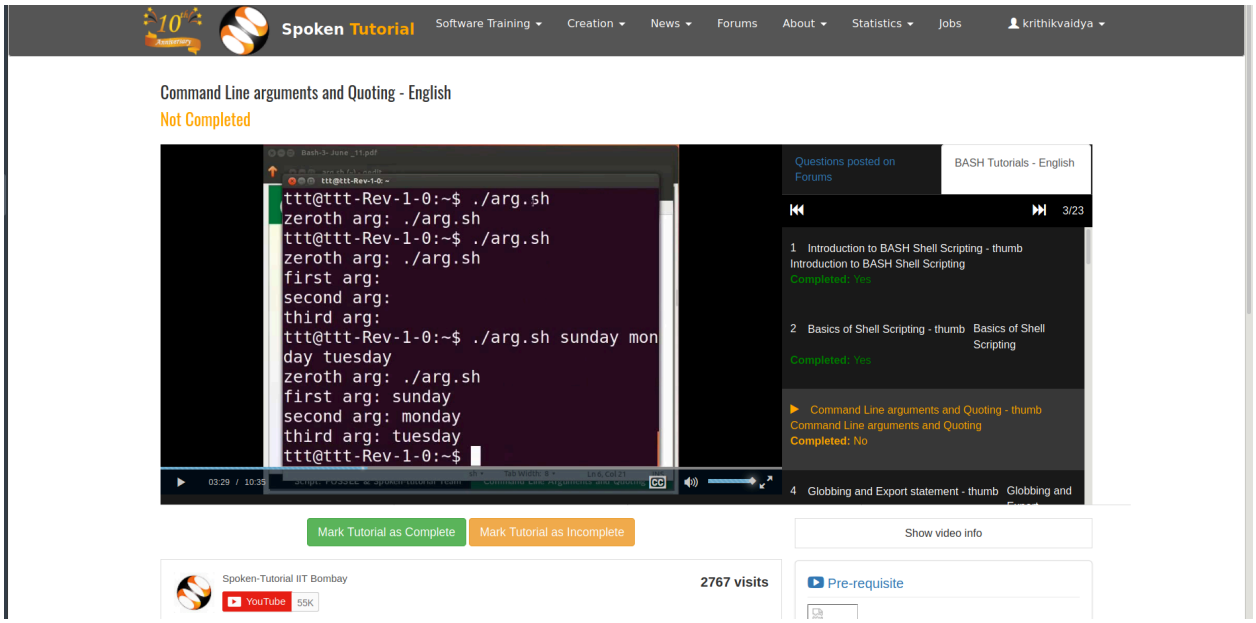
The setup accounts for saving logs in the scenarios of skipping ahead, fast-forwarding, going back, etc.

We can choose to update the tutorial progress at time intervals smaller than a minute, however, this will increase the load on the server.

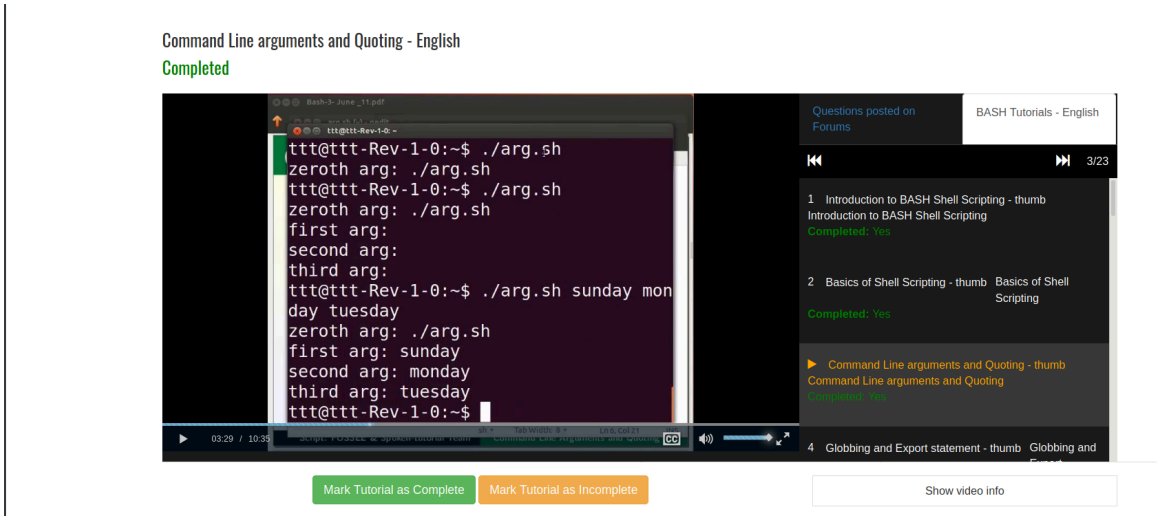
Please check the screenshots below (the formatting of the pages may look off in some places since the static files (images, thumbnails, etc) are not present on my local setup):



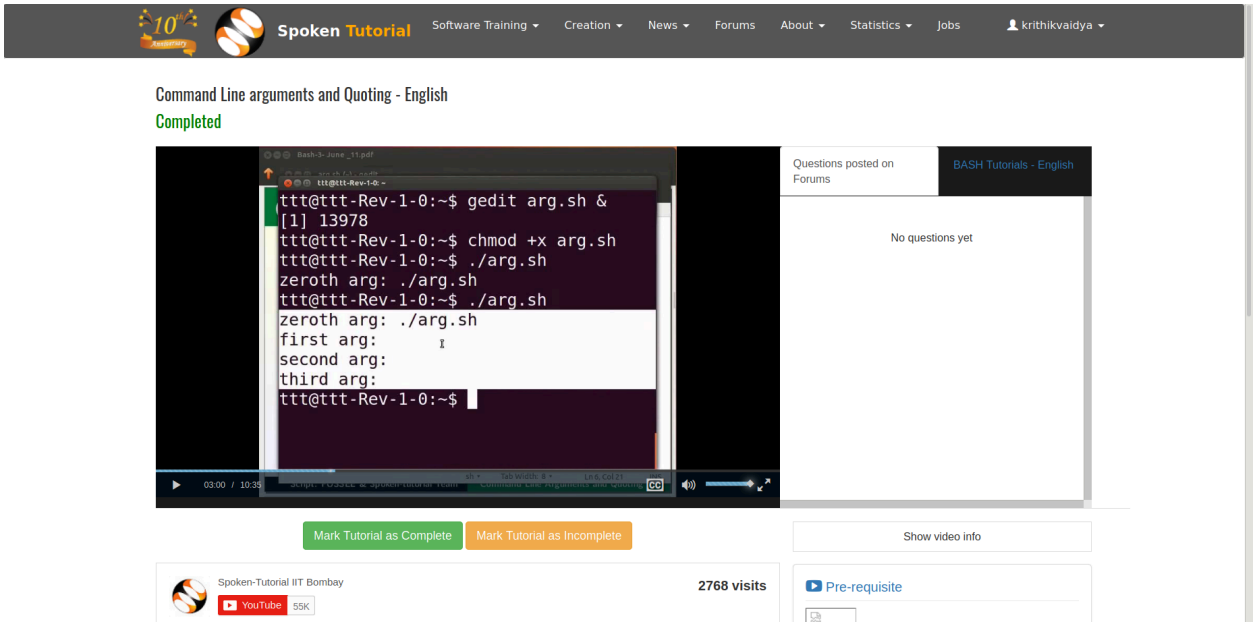
Showing tutorial completion status, buttons to mark as complete/incomplete



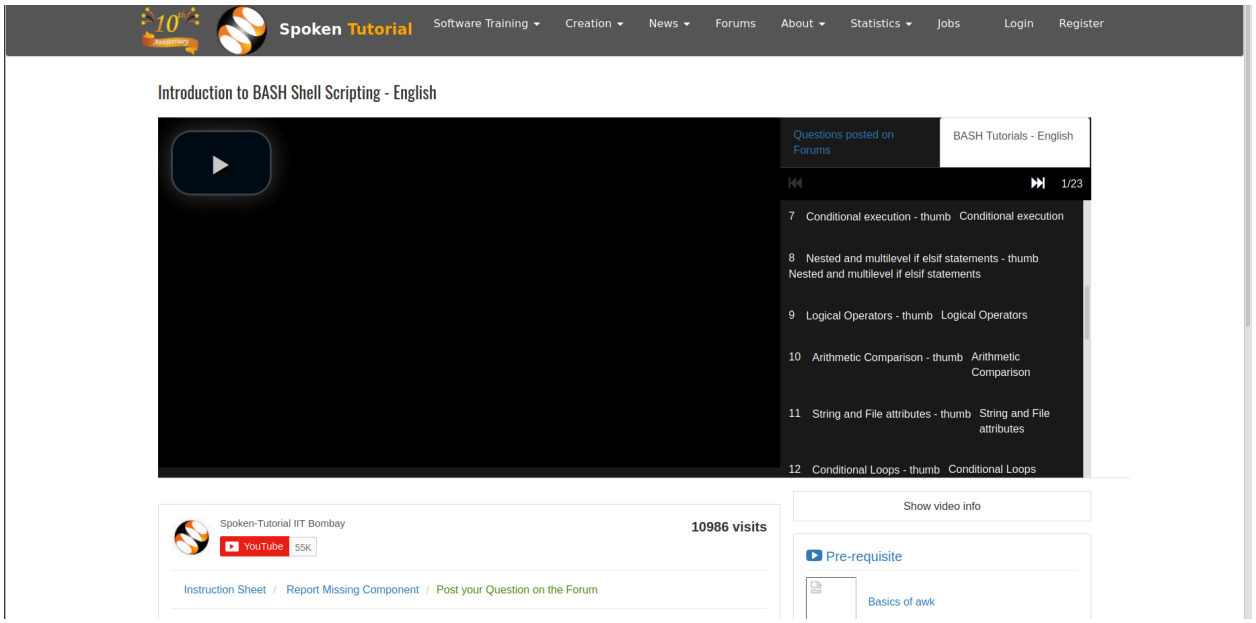
The tutorial completion status is also visible in the playlist, on the watch tutorial page



Clicking the Mark Tutorial as Complete button will make an AJAX call to an API. This API updates the “completed” field of the correct MongoDB document as ‘true’. After this succeeds, the completion status at the top of the page and in the playlist is automatically updated



If we close the video between 3:00 and 3:59, the next time we visit it, it continues at 3:00 minute mark

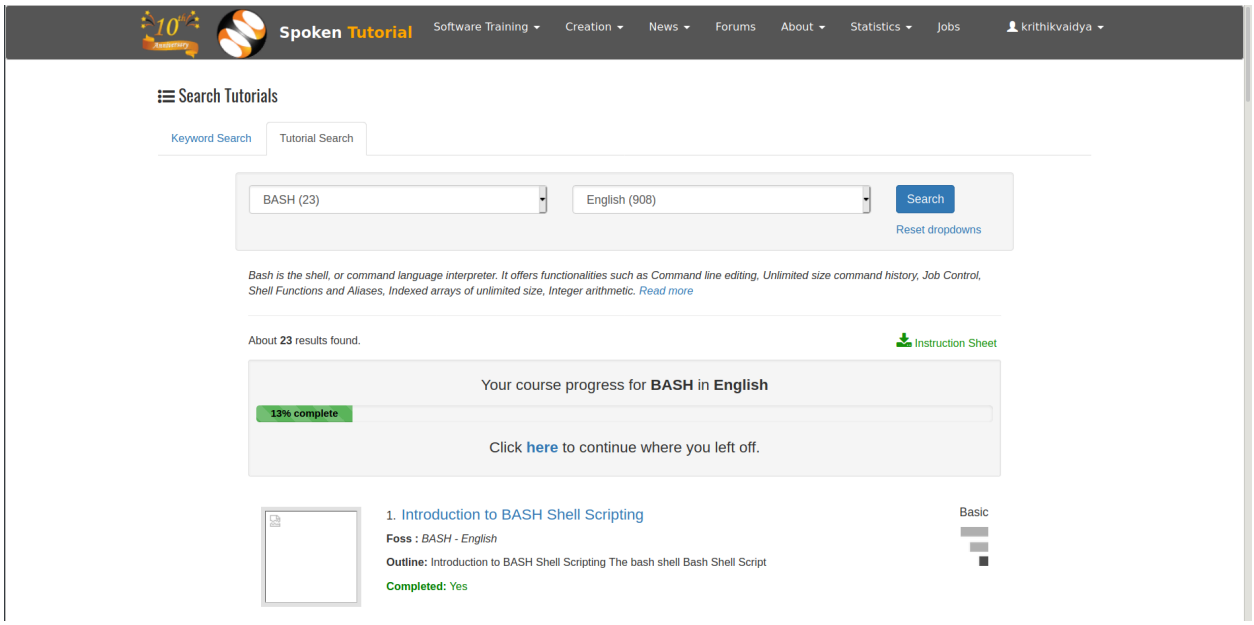


For non-logged-in users - none of the progress data is saved or visible

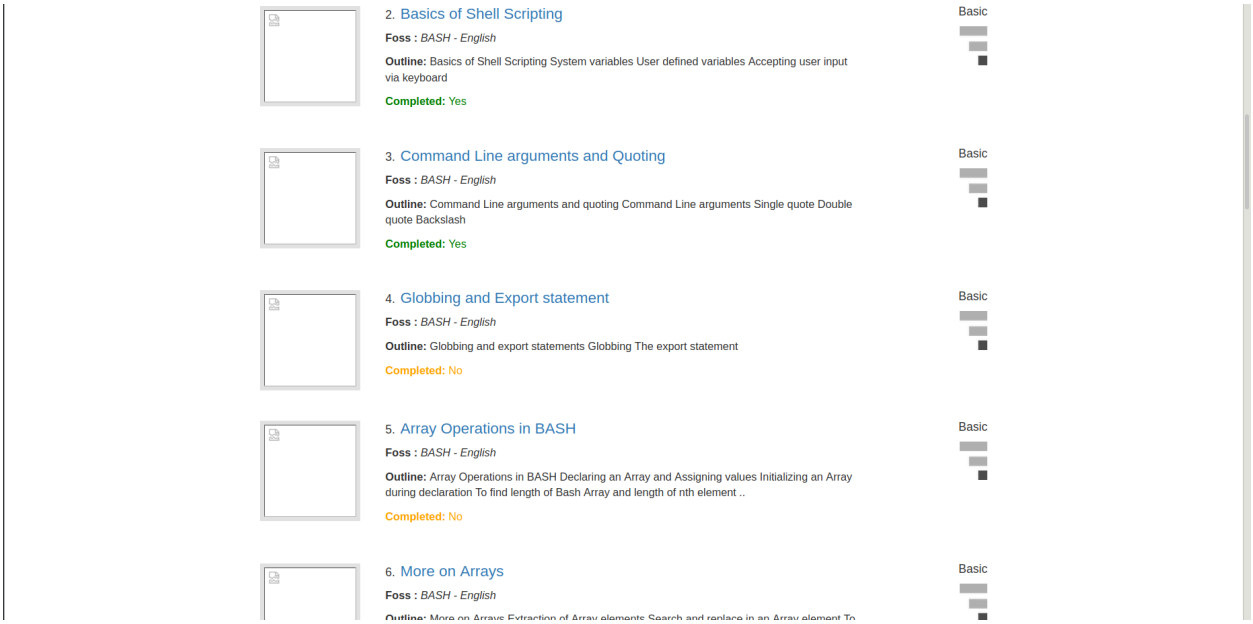
Once 80% of the minutes of the video are completed, the tutorial is automatically marked as completed (if it wasn't already marked) in the database, and the webpage is automatically updated to reflect the new completion status.

Tutorial Search page

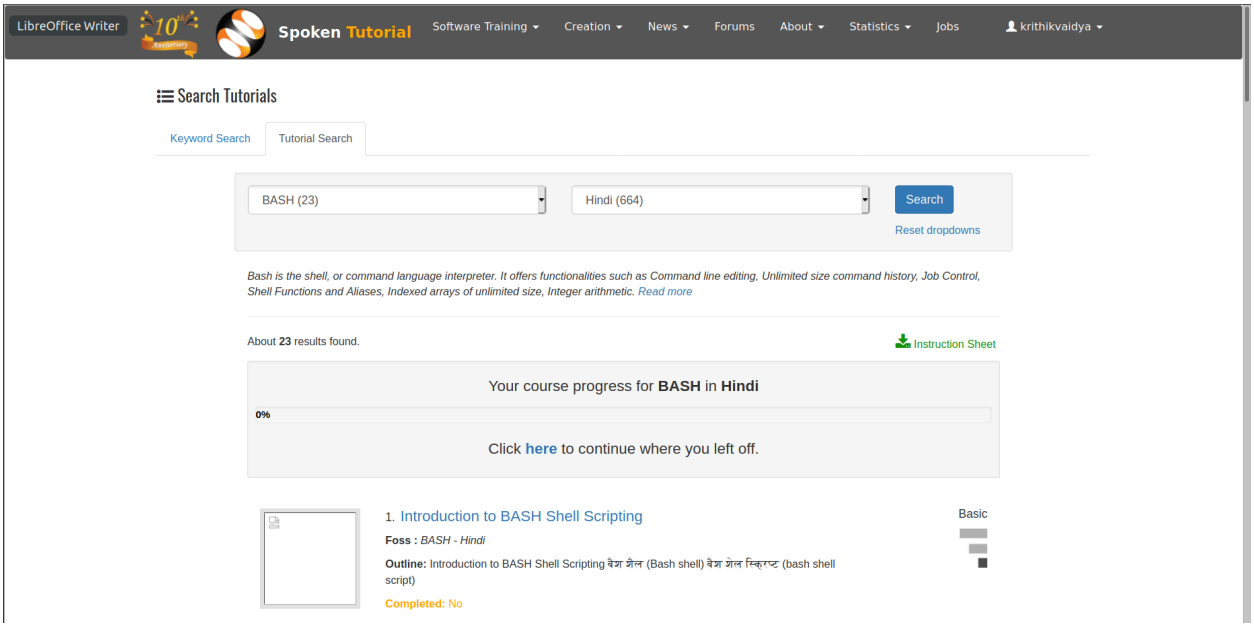
On the tutorial search pages, such as [this one], when the user is authenticated and has chosen a FOSS and a language, the course completion percentage and the option to continue watching where they stopped watching is given, as shown below.



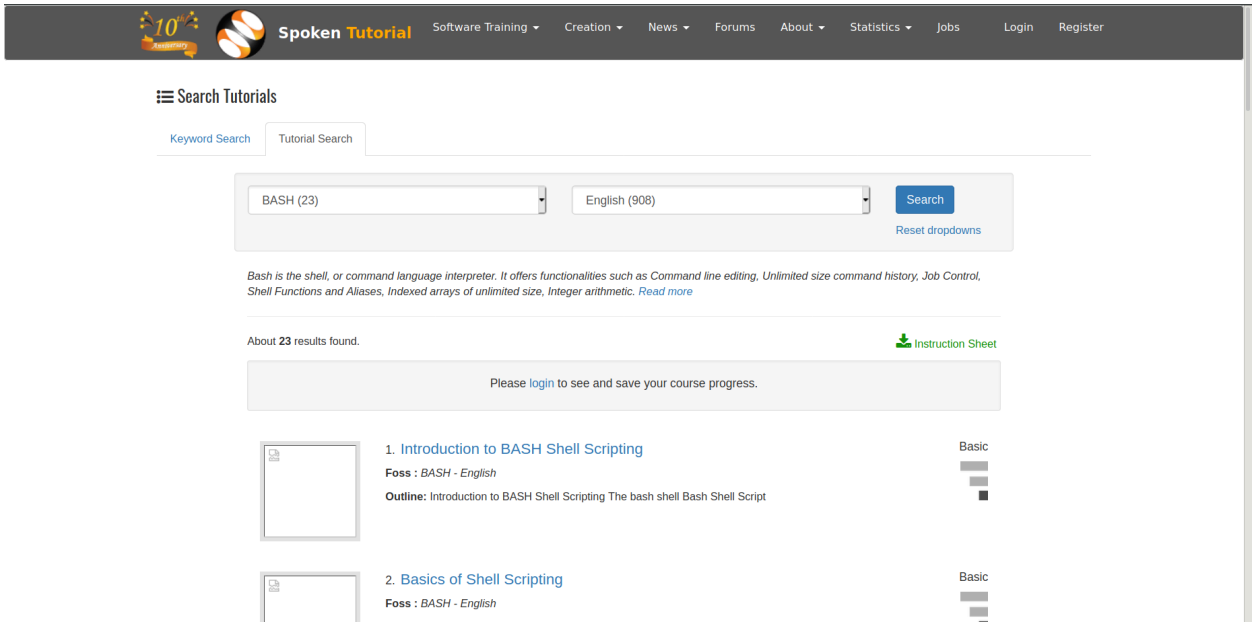
Course completion status, continue where you left off



The completion status for each tutorial is also shown, as can be seen.



Progress is recorded separately for different languages of the same FOSS



None of these things are visible to non-authenticated users

Schema for saving tutorial progress logs

Currently, the tutorial progress logs are being saved in the same database as the website logs (*logs* database), but in a different collection (*tutorial_progress_logs* collection).

The below describes an example of one document in the collection:

https://gist.github.com/krithikvaidya/25662dca3e80e0d7e59a253de71ba1a3

The logged data can also be used for statistical calculations and visualizations.

Other Explorations

Visit Duration Logging

To log the visit duration along with all the other visit data in a single log, we need to change the structure of the Javascript code. Instead of extracting the data and saving the logs on DOMContentLoaded, we will only extract the data on DOMContentLoaded, and save the logs before the page unloads, so that the duration of the visit can also be calculated. To do this, we can use window.unbeforeunload or jQuery's beforeunload functionality. For mobile users, when the page focus is lost (tab changed, browser closed, etc.), marks the end of the visit.

However, when tested with different browsers, these features were found to behave inconsistently and unreliably. Another option is the `navigator.sendBeacon` method which allows us to asynchronously send a small amount of data over HTTP to a web server. However, older browsers do not support this feature, and this extensive study about the feature concluded that navigator.sendBeacon is broken, and should not be used in production.

The code for this attempted approach can be found here.

Redis Persistence

In order to make sure that Redis queue data isn't lost during restarts/crashes of the system. There are two ways of doing this:

The Redis Database Backup

The Redis Database Backup, or RDB, files are snapshots that are taken at predetermined frequencies, to be used as a backup in a point-in-time recovery in the event of a data storage failure.

The Append-Only File

The append-only file, or AOF, is a mode of data persistence where Redis persist the dataset by taking a snapshot and then appending the snapshot with changes as those changes take place.

These methods can be used together, separately, or not at all in some circumstances.
For our use case, the AOF persistence method is more appropriate.
In basic terms, append-only log files keep a record of data changes that occur by writing each change to the end of the file. In doing this, anyone could recover the entire dataset by replaying the append-only log from the beginning to the end.

```
appendonly no
appendfsync everysec
no-appendfsync-on-rewrite no
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
```

Table 4.1 - Sync options to use with appendfsync

Option	How often syncing will occur
always	Every write command to Redis results in a write to disk. This slows Redis down substantially if used.
everysec	Once per second, explicitly syncs write commands to disk.
no	Lets the operating system control syncing to disk.

In our case, using appendfsync always would be the most ideal. However, if we were to set appendfsync always, every write to Redis would result in a write to disk, and we can ensure minimal data loss if Redis were to crash. Unfortunately, because we're writing to disk with

every write to Redis, we're limited by disk performance, which is roughly 200 writes/second for a spinning disk, and maybe a few tens of thousands for an SSD (a solid-state drive).

As a reasonable compromise between keeping data safe and keeping our write performance high, we can also set appendfsync everysec.

A sample Redis configuration file with appendfsync can be found in the repo here.

Further Reading

MongoDB-Elasticsearch Connection and Sync

For improved visualizations and performance of statistics calculations, moving the logs from MongoDB to Elasticsearch was briefly explored. To sync a MongoDB collection to an Elasticsearch collection, we explored the following options:

Initial choices:

- Mongo Connector
- Mongoosastic
- Mongo Stream
- Monstache
- Logstash Input MongoDB
- MongoDB input driver plugin for logstash

After considering various factors like appropriateness for our use case, frequency of updates, presence of support, ease of use, support for newer versions of Elasticsearch etc., we narrowed it down to two choices:

1. MongoDB input driver plugin for logstash

This uses the Logstash part of the Elastic stack to sync the two databases. Since Logstash does not have an official MongoDB input plugin, we have to use a 3rd party plugin. The

logstash configuration file for this purpose can be found in the Spoken-Analytics-System repository here.

To understand the use of the driver, please refer to the following links:

- Link 1
- Link 2
- Link 3
- Link 4
- Link 5

2. Monstache

Monstache is a go daemon that syncs MongoDB to Elasticsearch in real-time. Probably the easiest to setup.

Saving of Offline Tutorial-Related Logs

A good portion of Spoken Tutorials users hail from rural-type areas, where internet connectivity is limited. In these cases, the cdcontent download feature is used to download the Spoken Tutorials and view them locally, in a web browser. To record these logs, we can use the HTML5 Web Storage API to save the required logs locally. When the user comes online, the browser can automatically send this data to a Spoken Tutorials API.

Conclusion

This project dealt with creating robust pipelines for the saving of user activity logs, such as website browsing logs and course/tutorial progress logs. This data can be then used for generating insights, analytics, and visualizations of the user activities on the website. The presence of tutorial progress logs and their associated features contribute toward improving the user experience. Further explorations had been done with respect to Redis data backup, visit duration logging, and pipelining data between MongoDB and Elasticsearch.

For future improvements, we can extend the events tracking to other subdomains under spoken-tutorials.org (such as forums.spoken-tutorial.org, process.spoken-tutorial.org, etc.) and improve the cdcontent download logging. We can also log the user activity paths, similar to how it is shown by StatCounter. The idea of syncing MongoDB and Elasticsearch can be further explored to improve the performance of statistical calculations and visualizations.

In the end, I would like to thank FOSSEE and the Spoken Tutorials project for providing me the opportunity to work on this project. I would like to thank my mentor, Sir Abhijit Bonik, for providing guidance and allowing the freedom to explore different approaches. I would also like to thank Ma'am Nancy Varkey and Ma'am Kirti Ambrey for periodically reviewing my work.